Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads

Shilei Tian$^{1[0000-0001-6468-6839]}$, Johannes Doerfert$^{2[0000-0001-7870-8963]}$, and Barbara Chapman$^{1[0000-0001-8449-8579]}$

[1] Department of Computer Science, Stony Brook University
{shilei.tian, barbara.chapman}@stonybrook.edu
[2] Argonne Leadership Computing Facility, Argonne National Laboratory
jdoerfert@anl.gov

Abstract. In this paper, we introduce a novel approach to support concurrent offloading for OpenMP tasks based on hidden helper threads. We contrast our design to alternative implementations and explain why the approach we have chosen provides the most consistent performance across a wide range of use cases. In addition to a theoretical discussion of the trade-offs, we detail our implementation in the LLVM compiler infrastructure. Finally, we provide evaluation results of four extreme offloading situations on the Summit supercomputer, showing that we achieve speedup of up to 6.7× over synchronous offloading, and provide comparable speedup to the commercial IBM XL C/C++ compiler.

Keywords: Runtime optimizations · GPU · Accelerator offloading

1 Introduction

Parallel programming is here to stay. In fact, the number of compute cores configured per platform continues to grow, and many of them are in the form of accelerators. GPUs are the most common type of accelerator in modern supercomputers; on some recent systems, multiple GPUs are present on a single node. As most of the computational power is within them, it is imperative for performance (per watt) to keep the GPUs occupied with productive work at all times. A meaningful approach is to perform as many computations as possible simultaneously [10,12]. Even NVIDIA Fermi GPUs, which have been on the market for ten years, allow for concurrent execution of up to 16 GPU kernels on a single device. Asynchronous offloading is a promising technique to achieve such concurrency as it allows a single CPU thread to overlap memory movement, GPU computation, and the preparation of new GPU tasks on the CPU. Costly stalls between GPU computations, aka. *kernels*, are avoided and the hardware can start the execution of an already prepared kernel as soon as the ones currently executed stop utilizing the entire device.

The OpenMP standard supports asynchronous offloading since version 4.5, though compiler support still varies. In OpenMP, computations are mapped to

```
#pragma omp target depend(...) map(...) ... nowait
{ ... }
```

Fig. 1: Generic `target` directive with `task` parts, e.g., the `depend` and `nowait` clause, offloading parts, e.g., the `map` clause, and other clauses such as `shared`.

accelerators via `target` directives such as the one sketched in Fig. 1. The statement following the directive is called the *target region* and the task created by the directive is called a *deferred target task*. Similar to other tasks, dependences can be specified with the `depend` clause and asynchronous execution can be permitted with the `nowait` clause. The `map` clause can be used to ensure memory regions are mapped between the host and the device. Depending on the situation and the clause arguments this can result in memory allocation, copies, deallocation, or none of these. While we describe the necessary semantics, we refer to the OpenMP standard for details and additional information.

In this paper we propose, compare, and evaluate a new scheme to implement the `nowait` clause on `target` directives to achieve concurrent offloading. It is designed to provide good performance regardless of the context. Our approach utilizes otherwise "hidden" helper threads to provide consistent results across various use cases. In Section 2.2 we introduce several possible implementations and compare them from a theoretical perspective. We discuss the implementation of our *hidden-helper-thread* design as part of the LLVM compiler in Section 3, before providing an evaluation of its behavior for four extreme offloading cases in Section 4. Our results show the *hidden-helper-thread* design gains up to 6.7× improvement on Summit supercomputer, and also provides comparable speedup to the commercial IBM XL C/C++ compiler. We discuss related work in Section 5 and conclude with ideas for improvement in Section 6.

2 Design Discussion

In order to discuss different implementation designs for *deferred target tasks*, we first dissect one and identify its semantic steps. As part of the overall strategy it is important to determine which thread will execute each step, as that is a fundamental property of the design. Based upon this mapping of responsibilities, it is possible to reason about the performance potential of a given scheme in various scenarios without implementing all schemes and evaluating all scenarios.

1. wait for outstanding dependences
2. copy requested memory to the device
3. execute the target region on the device
4. copy requested memory from the device
5. resolve outgoing dependences

Fig. 2: Breakdown of the semantic parts, or sub-tasks, of a *deferred target task*.

The steps taken to execute a *deferred target task* are shown in Fig. 2. The first step is to resolve outstanding dependences, that is, wait for completion of previously generated sibling tasks that the target task depends on. Next, the memory regions are copied from the current, or issuing, device to the target device as specified by the `map` clauses. In step three, the associated target region is executed on the target device [3]. Afterwards, memory is copied back from the target device to the issuing device, again as specified by the `map` clauses. Finally, dependences are marked as resolved such that dependent tasks are now allowed to proceed. While these semantic steps could potentially be overlapped, they have to appear *as-if* they are performed in this order.

2.1 Considered Designs

We considered three designs that we describe here and compare in Section 2.2.

Regular Task In the *regular-task* design, the "task part" of the `target` directive is executed as if it was a regular, *undeferred* OpenMP task. A potential lowering of the generic *deferred target task* from Fig. 1 is shown in Fig. 3. As with other regular OpenMP tasks there is a binding to the encountering team, so that a thread from the encountering team will eventually execute the task. That thread will execute all five of the steps listed in Fig. 2, thus allowing the encountering thread to continue execution immediately after creating the regular OpenMP task. The *regular task* design is the easiest to implement and understand. However, it may not yield the desired result, namely asynchronous offloading, if there is no surrounding parallel region, or if the threads in the surrounding parallel region are busy and not able to pick up additional tasks.

```
#pragma omp task shared(...) depend(...) ...
#pragma omp target map(...) ...
{ ... }
```

Fig. 3: The `target` directive from Fig. 1 implemented in the *regular-task* design. The "task part" of the *deferred target task* becomes a regular, *undeferred* task.

Detachable/Callback-Task The *detachable-task* design exploits semantics similar to the `detach` clause in combination with asynchronous calls to the native device runtime. Fig. 4 visualizes this approach using a custom *native_async* clause. The idea is that the native device runtime, e.g., the CUDA driver, allows the queuing of events, memory copies, and launch kernels. The encountering thread can therefore set the first four steps shown in Fig. 2 in motion without waiting for any of them to complete. In practice, the fifth step can also be scheduled by providing a callback function for the native runtime to invoke once all prior steps have completed. The callback will fulfill the event associated with the `detach` clause and thereby, most likely, also perform the work associated with resolving dependences. That means that the encountering thread issues the work, and a thread of the native runtime will handle everything else, especially the last step. Consequently, if the native runtime is rich enough and has sufficient threads to perform the (last) step, concurrent offloading is possible regardless of the context.

[3]The fallback case, execution on the issuing device, is sufficiently similar.

```
#pragma omp target depend(...) map(...) ... \
                   detach(native_async_calls_done) native_async
{ ... }
```

Fig. 4: The `target` directive from Fig. 1 implemented in the *detachable-task* design. Asynchronous calls to the native device runtime are used to issue the subtasks (see Fig. 2) including a host callback that will fulfill the *allow-completion-event* associated with the `detach` clause.

Hidden Helper Task In the *hidden-helper-task* design, a *deferred target task* is executed in its entirety by a thread that is not started by nor (in any language-defined way) visible to the user. These *hidden-helper-threads* form a team of threads that is implicitly created at program start and is only responsible for the execution of the special *hidden-helper-tasks*. We denote them in our code as `hht_task`. Such tasks are not too different from other *deferred* OpenMP tasks except that they are always executed by an implicit *hidden-helper-thread*. It is especially important that they participate in the dependence resolution like any other tasks generated by the encountering thread. Thus they are siblings to tasks generated by threads in the same team as the encountering thread. The `hht_task` concept is not tied to *deferred target tasks* but could help the definition or extention of the OpenMP specification (see Section 6). Fig. 3 shows how the generic *deferred target task* from part 1 is executed in this design.

```
#pragma omp hht_task shared(...) depend(...) ...
#pragma omp target map(...) ...
{ ... }
```

Fig. 5: Conceptual lowering of the `target` directive from Fig. 1 in the *hidden-helper-thread* design. A special `hht_task` is used and executed by an *hidden-helper-thread* while the offload part is made synchronous.

2.2 Design Comparison

While all three schemes can result in concurrent execution of asynchronous offloading regions, they differ in complexity, extensibility, requirements, and probably performance. The regular task design is easy to implement, potentially even without compiler support, but it will fail to achieve the goal if there are no threads available to perform the offloading concurrently. Under ideal circumstances it can be expected that this scheme is similar to the design of *hidden-helper-thread*, though the required setup, e.g., an explicit parallel region with idle threads, is unrealistic and restrictive. The detachable task design can be expected to provide consistently good results under most circumstances. It could potentially be worse than the hidden-helper task design if the time taken by the encountering thread to issue asynchronous calls becomes the bottleneck or if the native runtime thread is otherwise needed while it resolves the dependences. However, those situations would only occur if the tasks are very small or the number of native runtime threads is too low. Moreover, the setup of the `hht_task` is not free either and the use of additional threads and task incurs overhead as well. There are more likely problems with the detachable task design though. For one, the scheme can become complex when dependences between host and target tasks are present. While one could resolve host task dependences as part of the setup, thus stalling the encountering thread until they are resolved, it would defeat the purpose. Using artificial host tasks to do the setup introduces the same problems as the regular task design; using extra threads is not much different from our proposed third design, but more complex for yet-to-be-determined gains. Finally, only the hidden-helper task design is generic and reusable. It puts no requirements on the native runtime, nor is the scheme tied to target offloading. That said, it is very likely that our scheme would benefit from resolving dependences directly on the device.

3 Implementation

To ensure concurrent execution of target tasks in every situation we need to augment the LLVM OpenMP runtime in two places: (1) we added *hidden-helper-threads* to which the execution of *target tasks* can be deferred, and (2) we utilize native device runtime features to offload multiple target tasks at the same time. In this section, we first introduce the key implementation details for hidden helper tasks and the hidden helper task team, before we discuss the support for concurrent task execution using multiple streams. Finally, we present the new dependence process mechanism.

3.1 Hidden Helper Task

In our design, a `target nowait` directive will be wrapped into a *hidden-helper-task*, which is a special OpenMP task that can only be executed by a *hidden-helper-thread*. In this section, we will introduce the allocation and synchronization of a *hidden-helper-task*. The execution will be discussed in Section 3.2.

Allocation When the *encountering thread* T_E reaches a hidden helper task t_h, it registers t_h as a child by incrementing the child task counter and it also increments the number of unfinished hidden helper tasks of its team. Then T_E enqueues t_h in the task queue of a hidden helper thread chosen based on T_E's global thread id. This selection ensures that hidden helper tasks are distributed evenly if they are encountered by multiple threads at the same time. Finally, T_E increases a semaphore S_H which we will discuss in Section 3.2. Once the task is finished, the children and unfinished task counters will be decreased by one.

Synchronization The synchronization of hidden helper tasks follows the rules of regular OpenMP tasks. They can be synchronized explicitly via a `taskwait` directive or implicitly at the end of a parallel region. For the explicit synchronization, the encountering thread T_E waits until the number of unfinished child tasks is zero. The implicit synchronization happens before the master thread of the team spawned by the `parallel` directive leaves the parallel region and continues execution of the succeeding statement. The master thread will first wait for all unfinished hidden helper tasks created by its team to complete.

3.2 Hidden Helper Thread Team

The *hidden helper thread team* is a special OpenMP team that, similar to the implicit initial team, exists at program start. It is not connected to the implicit initial team. The size of the hidden helper thread team, denoted by N_H, defaults to 8 in our implementation. It can be configured via an environment variable. This might be necessary based on the kernel sizes and hardware capabilities, e.g., if $N_H = 8$ threads fail to offload sufficient work while there is more available, the size should be increased. Just as with a regular team, the hidden helper thread team is implemented using a fork-join model. To avoid overheads when the feature is not used, the team is only initialized when the first hidden helper task is encountered.

The encountering thread T_E first creates a new thread T_H using the native host threading API. Note that T_H is not related to any other regular OpenMP threads but is similar to the initial thread that exists at the start of program execution. We call thread T_H the *master thread* of the hidden helper thread team. This new thread creates $N_H - 1$ hidden helper threads using the same facilities that other newly created OpenMP teams would use. That means, the T_H is not connected to the existing team structure but the hidden helper thread team is itself a regular OpenMP team. T_E is allowed to proceed only after the new team has been initialized and is ready to accept tasks.

While the team behaves like a regular one, the hidden helper threads are set up slightly differently from regular OpenMP threads. A regular OpenMP worker thread (in the LLVM OpenMP runtime) keeps looping with the expectation that one hardware thread is allocated to it. It is optimized for fast reaction time, so once a regular OpenMP task is encountered by its team the worker thread can pick it up and execute it right away. In contrast, it is crucial that hidden helper threads do not occupy host resources if they are not used. Assuming the host is fully utilized by regular OpenMP threads, there are no CPU cycles left for the hidden helper threads to use. In order to avoid contention, the hidden helper threads immediately block on the semaphore S_H after their setup is complete. Whenever a new hidden helper task is enqueued, S_H is incremented and at least one hidden helper thread is woken up to execute the task. After the execution is finished, the thread will block itself on S_H and wait to be woken again.

Like regular OpenMP threads, hidden helper threads use a work-stealing strategy to find suitable tasks. A hidden helper thread first checks whether there are tasks in its own queue. If so, it will take one and execute it; otherwise, the thread will try to steal from other thread queues by sweeping over all others in its hidden helper thread team.

3.3 Stream Manager

Steps 2–4 in Fig. 2 show how the host interacts with a target device via the native device runtime. These runtimes usually accept a queue-like data structure that we call a *stream*[4] in this paper, as a parameter to which the corresponding operations are pushed. Operations in the same stream are executed in the issued order; operations in different streams can be executed concurrently if there are no synchronizing events. As a result, if we want to run multiple tasks concurrently, we must use *multiple streams*.

We implemented a *stream manager* which can efficiently arbitrate concurrent requests for streams. Initially, a stream pool containing K streams is created. The size K is configurable via an environment variable and defaults to 32. For each target device operation, e.g., a host to device memory copy or a kernel offload, a new stream is requested from the stream manager. On request, the last used available streams in the pool can be "borrowed". If all streams in the pool have been borrowed, the stream manager will double the pool size to create fresh streams that can be handed out. Once a user is finished with a stream, it returns it to the stream manager such that it can be reused.

Since the target region can be only executed after the required data is copied to the device, and outgoing dependence resolution can only be started after data is copied back to the issuing device, Steps 2–4 in Fig. 2 are in fact implicitly dependent. Given this fact, we optimize the target operations in the following way: all operations for the same target task use the same stream. In addition, all synchronous operations are replaced by their asynchronous counterparts, with a single synchronization performed at the end of Step 4. In this way, the OpenMP runtime library does not need to wait for an operation to finish before issuing the next one. But it will register them all directly with the native runtime, allowing for potential concurrency during memory transfers in the future. Even for non-detachable target tasks, this synchronization scheme can reduce overheads compared with the use of multiple synchronous operations. Finally, it is worth noting that the stream manager alone already allows multiple threads to concurrently offload independent operations.

3.4 Processing Dependences

The dependences of a regular OpenMP task are resolved and processed on the host side. If the dependences of a task are not fulfilled, the task will not be enqueued, which implies that a target task will also not be enqueued, dispatched and executed if the tasks it depends on are not finished, no matter whether they are regular tasks or target tasks. However, almost all device runtime libraries support a more efficient way to process dependences via device-dependent *event*s. The host side no longer is involved, and all a target task's successors whose dependences have been resolved can be enqueued for dispatch and execution.

[4]This is CUDA terminology, but almost all heterogeneous programming models have a similar concept, such as the *command queue* in OpenCL.

The native device dependence resolution works as follows. A *fulfill* operation is put into the stream S such that it is executed after all operations enqueued to S before. A *wait* operation is added to stream S', which can be S, to ensure operations enqueued into S' afterward are stalled until the matching fulfill operation in S was executed. It is worth noting that the fulfill and wait operations are put into the stream without blocking the issuing thread.

In our approach, we perform dependence processing on the target device. Assume a target task t depending on m tasks $\{t_{d_1}, \cdots, t_{d_m}\}$. For each task t_{d_i}:

– If t_{d_i} is a regular task: add t to t_{d_i}'s *successor* list, and increment t's counter of predecessors `npredecessors`. This is same as the existing mechanism.
– If t_{d_i} is a target task: add t_{d_i} to t's *predecessor* list.

In this way, if the `npredecessors` of a task is not zero, the task depends on unfinished regular tasks. All tasks in the predecessor list are target tasks and will be processed on the device side.

When t is started, it first checks whether its `npredecessors` is zero. If yes, the current task yields because target device events can not tackle dependences on regular tasks. After that, for each task t' in t's predecessor list, if t and t' are on the same *type* of devices, insert a wait for t''s event to t's stream. This approach does not work if t and t' are not on the same *type* of devices. Two target tasks are not on the same *type* of devices if they are not using the same set of device runtime interfaces. In that case, t will take a check-and-yield: check the status of t''s event; if the event is not fulfilled, t will yield its execution. After all dependences are processed, t proceeds to its remaining offloading work, such as data mapping and kernel launch.

The *hidden-helper-thread* executing *hidden-helper-tasks* will wait for the target parts (Step 2-4 in Fig. 2) before proceeding to Step 5 to make sure that this dependence process can also work when a host task depends on a target task.

4 Evaluation

We performed experiments with four synthetic benchmarks described in the following to show performance gained by asynchronous offloading with Hidden Helper Threads (HHT) over vanilla LLVM. We additionally compare the prototype to the implementation in the commercial IBM XL C/C++ compiler (XLC) by measuring the speedup of asynchronous offloading (with `nowait`) over synchronous offloading (without `nowait`).

4.1 Benchmarks

The benchmark functions `B1`, `B2`, `B3`, and `B4` contain the timed code parts. `B1`, `B2`, and `B3` each consists of a `target nowait` directive, and `B4` consists of four `target nowait` directives with `depend` clauses. In the target region a *daxpy*-like computation is performed on vectors of length `N`, as shown below. Outer data mapping, which is not shown in the paper, is used such that data is transferred only once in each benchmark. The benchmarks are designed to be extreme instances of potential real world situations.

```
#define C(a, X, Y, N)                         \
  for (int i = 0; i < N; ++i)                 \
    for (int j = 0; j <= i; ++j)              \
      y[i] = y[i] + a * x[j];

inline void K(double a, double *X, double *Y, int N) {
#pragma omp target teams distribute parallel for simd nowait
  C(a, X, Y, N);
}
```

B1: Single-threaded asynchronous offloading, no parallel region

In benchmark `B1`, a single thread issues the `T` asynchronous offloading requests before it waits for all of them to finish. A situation like this can arise if an application with independent parallel loops is ported to an accelerator. Existing `omp parellel for simd` are replaced with the `omp target teams distribute parallel for simd nowait` pragma.

```
void B1(double a, double *X, double *Y, int T, int N) {
  for (int t = 0; t < T; ++t)
    K(a, X, Y, N);
#pragma omp taskwait
}
```

B2: Multi-threaded asynchronous offloading inside a parallel region

In benchmark `B2`, all threads created by an outer parallel region issue the `T` asynchronous offloading requests. Note that the encountering thread can only finish the parallel region once all offloading requests have completed. One can imagine an implicit `omp taskwait` at the end of the parallel region. A situation like this can arise if an application utilizes multiple host threads for offloading onto the same (set of) devices.

```
void B2(double a, double *X, double *Y, int T, int N) {
#pragma omp parallel for
  for (int t = 0; t < T; ++t)
    K(a, X, Y, N);
}
```

B3: Single-threaded asynchronous offloading inside a parallel region

In benchmark `B3`, the master thread of an outer parallel region issues the `T` asynchronous offloading requests. Note that the other threads created by the parallel region do not participate in the offloading and will be busy (waiting) until all offloading requests have finished. A situation like this can arise if an application utilizes some host threads for offloading while others perform unrelated tasks, e.g., work on the host or offloading to other devices.

```
void B3(double a, double *X, double *Y, int T, int N) {
  std::atomic_bool done(false);
#pragma omp parallel
  {
    if (omp_get_thread_num() == 0) {
      B1(a, X, Y, T, N);
      done.store(true);
    } else {
      while (!done.load())
        ;
    }
  }
}
```

B4: Single-threaded asynchronous offloading with dependences

In benchmark `B4`, a single thread issues four asynchronous offloading tasks in each loop iteration. Task 2 and 3 depend on task 1, and task 4 depends on task 2 and 3. Task 2 and 3 are mutually independent, therefore they can be running concurrently. A situation like this can arise if an application with parallel loops is ported to an accelerator and multiple such loops are offloaded concurrently to improve the overall performance.

```
#define TARGET_NOWAIT \
 #pragma omp target teams distribute parallel for simd nowait

void B4(double a, double *X, double *Y, int T, int N) {
  for (int t = 0; t < T; ++t) {
// Hidden helper task 1
TARGET_NOWAIT depend(in: x[0:N]) depend(inout: y[0:N])
    C(a, X, Y, N);
// Hidden helper task 2, depending on task 1
TARGET_NOWAIT depend(inout: x[0:N])
    C(a, X, X, N);
// Hidden helper task 3, depending on task 1
TARGET_NOWAIT depend(inout: y[0:N])
    C(a, Y, Y, N);
// Hidden helper task 4, depending on task 2 and 3
TARGET_NOWAIT depend(in: x[0:N]) depend(inout: y[0:N])
    C(a, X, Y, N);
  }
#pragma omp taskwait
}
```

4.2 Configurations

We run our experiments with 13 different vector sizes, and four different number of offloading jobs. The vector size, determined by `N`, is one of 2^4, 2^5-1, 2^6, 2^7-1, 2^8, 2^9-1, 2^{10}, $2^{11}-1$, 2^{12}, $2^{13}-1$, 2^{14}, $2^{15}-1$, and 2^{16}. We choose seven powers of two as well as six values between them. The number of offloading jobs, determined by `T`, is one of 2^4, 2^6, 2^8, and 2^{10}.

4.3 Systems and Versions

All experiments were executed on the Summit supercomputer at Oak Ridge National Laboratory [2]. Each Summit node contains two IBM POWER9 processors and six NVIDIA Volta V100 GPUs.

We implemented out prototype on top of LLVM `d8c35031`. Since parts of our work have already been merged into the trunk of LLVM, in order to demonstrate our complete approach, the vanilla LLVM was obtained by removing the related changes from `d8c35031`. All variants of LLVM were built with GCC 7.4.0. For comparison, we use IBM XL C/C++ V16.1.1 (5725-C73, 5765-J13) loaded by default on Summit. CUDA 10.1.243 was used by all configurations.

All benchmarks were compiled with flags `-std=c++14 -O2`. We used one resource set, which has 42 CPU cores and one GPU, per execution. Each configuration was executed 30 times and the execution times were averaged. We ran the experiments using the following command:

```
jsrun --smpiargs="-disable_gpu_hooks" --nrs=1          \
      --tasks_per_rs=1 --cpu_per_rs=42 --gpu_per_rs=1 \
      --rs_per_host=1 --bind=rs PROGRAM
```

4.4 Results

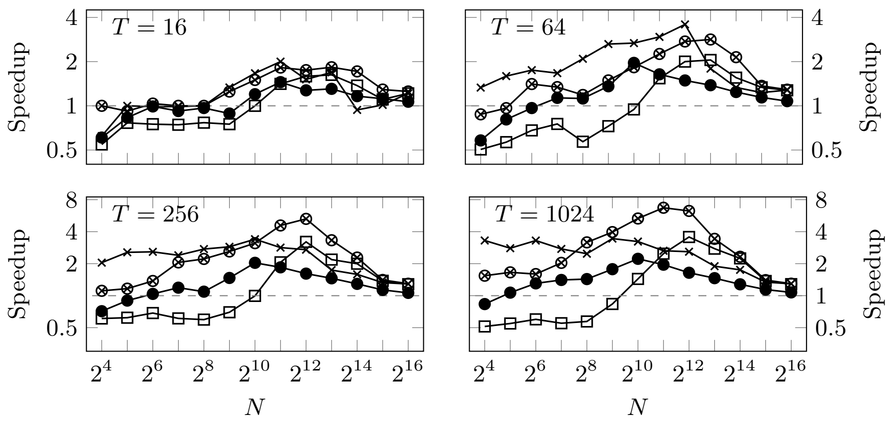

Fig. 6: Speedup of concurrent execution with *hidden-helper-threads* compared to vanilla LLVM for the benchmarks `B1` (⊗), `B2` (□), `B3` (×) and `B4` (●) described in Section 4.1

Comparison with Vanilla LLVM Fig. 6 shows the speedup of concurrent execution with our implementation. We can see in all cases that speedup first increases with vector length N (kernel size), starts to decrease after a certain point, and finally levels off. This is to be expected, because at the beginning when N is small, multiple concurrent target tasks cannot fully utilize the GPU. The extra overhead of the target tasks cancels out the small improvement in execution time. As N grows, we start to observe the improvement in execution time resulting from overlapping execution. At the point when a single target tasks saturates the GPU alone, the speedup decreases with the ratio of the target task execution time that is executed concurrently with other target tasks. Given large enough target tasks a speedup of 1 is expected.

We also note that the maximum speedup increases with T (number of target tasks). With increasing values of T the amount of time spent in the less concurrent warm-up and tear-down stages of the pipeline decreases relatively to the overall execution time, thus allowing for larger speedups.

Both `B1` and `B3` show significant performance improvement (up to 6.7×) in all configurations, while `B2` and `B4` exhibit degradation for small tasks and minor gains for larger values of N. This is to be expected, because `B2` contains multi-threaded offloading inside a parallel region. Even without the `nowait` clause, there are 168 threads (on Summit, each physical core supports four hardware threads) issuing offload requests almost at the same time. The implementation of the `nowait` clause introduces indirection through the *hidden-helper-thread*, which increases host-side overheads (for task allocation, scheduling, and task yield) that are not incurred when issuing multiple tasks directly from different native OpenMP threads. Here, the performance gain from introducing multiple streams is offset by the extra overheads of using a deferred target task when the kernel size is very small. In this case, the modest amount of time spent in kernel execution does not allow us to benefit from dispatching them into multiple streams. For `B4`, there are at most two concurrent tasks (2 and 3), and they can finish before they get a chance to run at the same time when N is very small. As a result, similar to `B2`, the extra overheads degrade performance.

Comparison with IBM XL C/C++ Compiler IBM XLC can generate very efficient kernels compared with LLVM (up to 50x performance gap with benchmarks in this paper), a lightweight configuration (smaller N) for XL can be heavy for LLVM. We are still exploring the reasons for this, but LLVM's register file usage on GPUs appears to be highly inefficient and could be a leading contributor. Since a direct comparison does not make sense given the above performance difference, we instead compare the speedup of pairs of kernels that have approximately the same average execution time.

Fig. 7 shows the speedup of asynchronous offloading over synchronous offloading using our prototype (HHT) and IBM XLC in different benchmarks (`B1-4`) and for different T (number of offloading jobs). We see that our HHT outperforms XL in `B2`, although there is performance degradation from both HHT and XL when the kernel execution time is short, as discussed above. However, when the kernel size is large enough, HHT improves performance considerably while XL cannot provide any improvement. We think this can be attributed to the use of hidden helper threads. For XL, we observe from the NVIDIA Visual Profiler (NVVP) [8], that when executing `B1` and `B3`, only one host thread is interacting

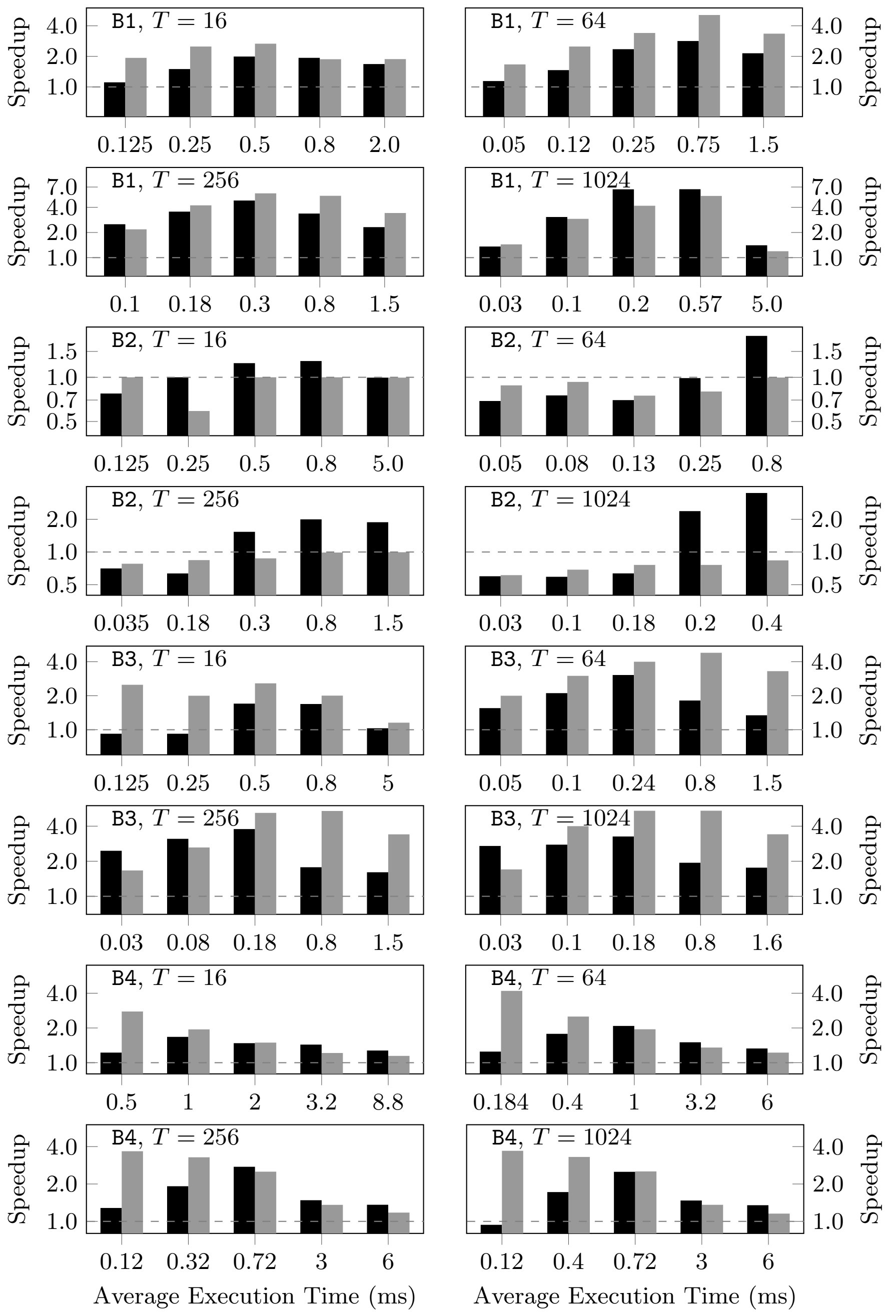

Fig. 7: Comparison between the speedup using our prototype (■) and IBM XLC (▨) in different benchmarks (B1-4) and T (number of offloading jobs). The average execution time is the total time of T target tasks executing serially divided by T.

with the CUDA runtime, which indicates that the offloading may be performed by the encountering threads. As a result, aside from the extra overhead introduced to support `nowait`, XL can at most get same performance as synchronous offloading in `B2`. In our approach, only eight *hidden-helper-threads* were issuing offloading requests, which has less resource contention and better cache locality compared with synchronous version.

For `B1`, `B3`, and `B4`, HHT can get speedup comparable with XL, even though generally it is slightly below that of XL. We expect this is due to weaknesses in LLVM's current kernel generation process. From the log shown from `ptxas`, which is the PTX assembler provided by NVIDIA [7], the kernel generated by IBM XL uses 32 registers, while the LLVM code uses 40 registers. The number of registers a thread block uses determines how many thread blocks can be resident on a multiprocessor [6]. In the LLVM version, fewer thread blocks will run on a multiprocessor at the same time, reducing the maximal concurrency.

Most important, our design provides more functionalities and it can be used even if the native runtime has no asynchronous offloading capabilities.

5 Related Work

OpenMP 4.0 provides mechanisms to offload regions of code to accelerators, and adds support for asynchronous offloading since version 4.5. Antao et al. [1] introduces an OpenMP offloading implementation to LLVM. For now it supports offloading to X86_64, AArch64, PPC64[LE], and has basic support for CUDA devices [3]. As regards support for the asynchronous offloading (`nowait` clause), Clang currently emits a corresponding function call but the function just calls the synchronous version, so that this feature is not supported. One key contribution of this paper is to propose a scheme to implement the `nowait` clause in LLVM.

GCC introduced support for offloading to the Intel® Xeon Phi™ from version 5, and support for the first GPU target, NVIDIA NVPTX, is introduced in GCC 7 [9]. Asynchronous offloading is not yet provided in GCC. It does not fall back to the synchronous version and therefore a program with `nowait` clause currently cannot run at all. Hence we were unable to provide a comparison. In the commercial space, the IBM® XL C/C++ V16.1.1 compiler fully supports OpenMP 4.5 [4], including asynchronous offloading.

Several papers investigate performance improvements by introducing concurrent offloading task/kernel execution with different programming models. Jiao et al. [5] validated the benefits of concurrent kernels for energy-efficient execution with CUDA. Wen et al. [11] proposed a graph-based algorithm to optimize OpenCL concurrent kernel execution. To the best our knowledge, this is the first paper investigating the concurrent execution of OpenMP target tasks.

6 Conclusions and Future Work

In this work we introduced support for concurrent execution of OpenMP target task, and discussed different designs for asynchronous offloading and evaluated our implementation on four extreme offloading situations against vanilla LLVM and IBM XL C/C++ compiler on the Summit supercomputer. Our results show that the *hidden-helper-thread* design can provide low-overhead, concurrent offloading of OpenMP target regions without support from the underlying native runtime. In addition, the proposed design can be a stepping stone towards other "free", "unshackled", or "non-team-bound" tasks, both in terms of implementation as well as design. Our next step is to improve the execution efficiency of kernels. A promising candidate approach is to reduce register usage in LLVM (see Section 4.4) by optimizing the device runtime library such that it can drop some parts if the kernel does not require those features.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

References

1. Antao, S.F., Bataev, A., Jacob, A.C., Bercea, G.T., Eichenberger, A.E., Rokos, G., Martineau, M., Jin, T., Ozen, G., Sura, Z., Chen, T., Sung, H., Bertolli, C., O'Brien, K.: Offloading support for openmp in clang and llvm. In: The Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). pp. 1–11. Salt Lake City, UT, USA (2016)
2. Facility, O.R.L.C.: Summit – oak ridge leadership computing facility, `https://www.olcf.ornl.gov/summit/`
3. Group, L.D.: Openmp support — clang 11 documentation – llvm, `https://clang.llvm.org/docs/OpenMPSupport.html`
4. IBM: Openmp support in xl c/c++, `https://www.ibm.com/support/knowledgecenter/SSXVZZ_16.1.1/com.ibm.xlcpp1611.lelinux.doc/getstart/omp_v1611.html`
5. Jiao, Q., Lu, M., Huynh, H.P., Mitra, T.: Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs. In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 1–11. IEEE, San Francisco, CA, USA (2015)
6. NVIDIA: Cuda c best practices guide, `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`
7. NVIDIA: Nvidia ptx optimizing assembler, `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html`
8. NVIDIA: Nvidia visual profiler, `https://developer.nvidia.com/nvidia-visual-profiler`
9. Project, G.: Offloading support in gcc, `https://gcc.gnu.org/wiki/Offloading`
10. Wang, L., Huang, M., El-Ghazawi, T.: Exploiting concurrent kernel execution on graphic processing units. In: International Conference on High Performance Computing & Simulation. pp. 24–32. IEEE, Istanbul, Turkey (July 2011)

11. Wen, Y., O'Boyle, M.F., Fensch, C.: Maxpair: Enhance opencl concurrent kernel execution by weighted maximum matching. In: Workshop on General Purpose GPUs. pp. 40–49. ACM, Vienna, Austria (2018)
12. Wende, F., Cordes, F., Steinke, T.: On improving the performance of multi-threaded cuda applications with concurrent kernel execution by kernel reordering. In: Symposium on Application Accelerators in High Performance Computing. pp. 74–83. IEEE, Chicago IL, USA (2012)